COP 4516: Math for Programming Contest Notes

Euclid's Algorithm

Euclid's Algorithm is the efficient way to determine the greatest common divisor between two integers. Given two positive integers a and b, we divide b into a, obtaining the remainder r, repeat the algorithm with b and r, finishing when r is 0. Here is a quick example that calculates the gcd of 144 and 57:

```
144 = 2 x 57 + 30
57  = 1 x 30 + 27
30  = 1 x 27  + 3
27  = 9 x 3
```

The gcd of the two original integers is the last non-zero remainder. For the example above, the greatest common divisor of 144 and 57 is 3, the remainder listed on the second to last line of the algorithm.

It's not too difficult to see that algorithm infers that the gcd(144, 57) = gcd(57, 30) = gcd(30, 27) = gcd(27, 3) = 3

Thus, we can code the gcd function using recursion quite succinctly:

```
public static int gcd(int a, int b) {
    return b == 0 ? a : gcd(b, a%b);
}
```

Mod Inverse

In many problems, it's useful to solve a modular equivalence equation, something like:

$$47x \equiv 13 \ (mod\ 85)$$

Normally, in a regular equation, we would divide through by 47. But this is not allowed in a modular equivalence. Rather, we must multiply the equation through by a value, c, such that 47c is equivalent to 1 mod 85, revealing just x. This value is defined as the modular inverse of 47 mod 85. Typically this is written as 47^{-1} mod 85. In general a^{-1} mod b exists if and only if gcd(a,b) = 1. More generally, if gcd(a,b) = 1, we can always find integers x and y such that:

$$ax + by = 1$$

To find a^{-1} mod b once we find values for x and y that satisfy the equation above, simply consider the equation mod b:

$$ax + by \equiv 1 \ (mod\ b)$$
$$ax \equiv 1 \ (mod\ b)$$

It follows from the definition of modular inverse that $a^{-1} \equiv x \ (mod\ b)$.

Luckily, when gcd(a, b) = 1, there is an extension to the Euclidean Algorithm that solves for a set of values (x, y) to satisfy the equation above. Here is the code for the Extended Euclidean Algorithm:

```
// Returns b inverse mod a in res[1].
// Note - value returned could be negative.
// Only works if a and b are ints due to the multiplication in the code.
public static long[] extendedEuclideanAlgorithm(long a, long b) {

    if (b==0)
        return new long[]{1,0,a};
    else {
        long q = a/b;
        long r = a%b;
        long[] rec = extendedEuclideanAlgorithm(b,r);
        return new long[]{rec[1], rec[0] - q*rec[1], rec[2]};
    }
}
```

One thing to note about any solution (x, y) to $ax + by = 1$ is that if (x, y) is a solution, so is any ordered pair of the form (x + bz, y - az), for any integer z. To see this, recall that our given information is that $ax + by = 1$. Now, consider plugging in x' = x + bz, y' = y - az, where z is any integer:

$$\begin{aligned} ax' + by' &= a(x + bz) + b(y - az) \\ &= ax + abz + by - abz \\ &= ax + by \\ &= 1 \end{aligned}$$

Thus, if we wanted to find all solutions to an equation of the form ax + by = 1, where a and b were given to us, we could find one solution using the Extended Euclidean algorithm and construct all other solutions by plugging in all possible integers for z.

Finally, if we have an equation of the form $ax + by = c$, where we want the general solution, we do the following:

1) if gcd(a, b) doesn't divide evenly into c, there is no solution, and we are done.

2) if gcd(a, b) isn't 1, then we can divide the whole equation through by the gcd and obtain a new equation where the gcd is 1. Let the three values we obtain by dividing by the gcd be a', b' and c', respectively.

3) Solve the equation as if 1 were the right hand side.

4) Multiply our solution (x, y) that solves the equation for 1 by c' to obtain a single valid solution (c'x, c'y).

5) Add and subtract multiples of a' and b' as necessary to obtain other solutions.

Prime Factorization

A prime number is a positive integer that isn't divisible by any integer but 1 and itself. The first few prime numbers are 2, 3, 5, 7, and 11. If an integer n is NOT prime (also known as composite), it can be written as a product of two positive integers, both greater than 1. Note that if xy = n with x ≤ y, since $\sqrt{n}\sqrt{n} = n$, it follows that $x \leq \sqrt{n}$. Thus, all composite numbers n have at least one divisor greater than 1 and less than or equal to the square root of n.

It follows that if we want to test a single number, n, for primality, it suffices to try to divide it (using % in code) by each possible divisor from 2 to $\sqrt{n}$.

Each integer has a unique prime factorization (the Fundamental Theorem of Arithmetic), namely, a list of primes that multiply to it. Typically, we write prime factorizations with each unique prime factor raised to the appropriate power. For example, $75 = 3 \times 5^2$.

Using the fact above, we can write code that prime factorizes an integer, n, in O($\sqrt{n}$) time. We simply start at 2 and try dividing into our integer n. Once we find a divisor, we "divide it out" and continue the process. Whenever we find a divisor, we must try that divisor again, just in case it divides into the original number more than once. Here is some code that prime factorizes an integer:

```
class pair {
    public int prime;
    public int exp;

    public pair(int p, int e) {
        prime = p;
        exp = e;
    }
}

In a class
----------
public static ArrayList<pair> primeFactorize(int n) {

    ArrayList<pair> res = new ArrayList<pair>();

    int div = 2;
    while (div*div <= n) {

        int exp = 0;
        while (n%div == 0) {
            n /= div;
            exp++;
        }

        if (exp > 0) res.add(new pair(div, exp));
        div++;
    }

    if (n > 1) res.add(new pair(n, 1));
    return res;

}
```

<u>Prime Sieve (Sieve of Eratosthenes)</u>

Now imagine that rather than needing to find if one number is prime, that we want to find all the primes from 2 to some positive integer n. A faster method than individually testing each integer exists, discovered by the Greek mathematician Eratosthenes. It works as follows:

1) Write down all the numbers from 2 to n.
2) Go through each number, in order.
3) For each of these, if it's not crossed off, circle it.
4) Then, cross off each multiple of that number. Thus, when we circle 2 at the beginning of the algorithm, then cross off 4, 6, 8, 10, and so forth, until we get to the last even number less than or equal to n.

In code, the following method returns a boolean array such that prime[i] is true if and only if i is prime:

```
public static boolean[] primeSieve(int n) {

    boolean[] isPrime = new boolean[n+1];
    Arrays.fill(isPrime, true);
    isPrime[0]= false;
    isPrime[1] = false;

    for (int i=2; i*i<=n; i++)
        for (int j=2*i; j<=n; j+=i)
            isPrime[j] = false;

    return isPrime;
}
```

We can optimize this in two ways:

1) Stopping the outer loop after we reach $\sqrt{n}$.

2) Skipping the j loop if isPrime[i] is already false. (For example, there is no point in crossing off multiples of 4 since we already crossed all of those off when we crossed off all multiples of 2 greater than 2.)

For nearly all contest questions, the code above runs fast enough for any list of primes we might need and there's no need to make the optimizations mentioned. If problems with very, very tight run time limits, these optimizations may be of use, so it's good to know them.

Number of Divisors of an Integer.

To determine the number of divisors of an integer, we can either use brute force (checking all divisors to the square root, realizing that each divisor strictly less than the square root of a number maps to a divisor greater than the square root), or if we have access to the prime factorized version of the integer, we can detemrine it almost immediately.

As an example, consider the integer $n = 2^4 \times 3^3 \times 5^2$.

Any divisor of it must have the form $2^a \times 3^b \times 5^c$, with $0 \le a \le 4$, $0 \le b \le 3$, and $0 \le c \le 2$.

Since any choice of a can be combined with any choice of b and c, it follows that we can simply multiply the number of possible choices for a, b and c to obtain the number of divisors of n. In this example, we have 5 x 4 x 3 = 60 divisors.

In general, if the exponents in the prime factorized form of an integer are $e_1, e_2, ..., e_k$, then the number of divisors of the integer is $(e_1 + 1)(e_2 + 1) ... (e_k + 1)$.

Note that one consequence of this formula is that the only way the number of divisors of an integer can be odd is if each of the factors above is odd. But that only occurs if all of the exponents in the prime factorization are even. If this is true, then the number is a perfect square. Thus, all integers EXCEPT perfect squares have an even number of divisors. The reason that perfect squares have an odd number of divisors is that each of their divisors can be written in pairs EXCEPT for the square root of the number. Take for example, 36. We can write its divisors in pairs that multiply to 36: (1, 36), (2, 18), (3, 12), and (4. 9). But, we would have to pair 6 with itself, but we don't want to count 6 twice!!! For all non-perfect squares, we always just list the pairs so the count must be even.

Sum of the Divisors of an Integer

Consider adding all of the divisors of $n = 2^4 \times 3^3 \times 5^2$. We know that all of them are of the form $2^a \times 3^b \times 5^c$. Now, consider the following multiplication:

$$(2^0 + 2^1 + 2^2 + 2^3 + 2^4)(3^0 + 3^1 + 3^2 + 3^3)(5^0 + 5^1 + 5^2)$$

Notice that every term foiled out is of the form $2^a \times 3^b \times 5^c$. Furthermore, notice that each unique divisor appears exactly once in this expansion. Thus, this product equals the sum of the divisors of n. Finally, since each of the sums within the parentheses are geometric series, we can plug into the formula for the sum of geometric series to get the sum of the divisors of n to be:

$$\frac{(2^5 - 1)}{(2 - 1)} \times \frac{(3^4 - 1)}{(3 - 1)} \times \frac{(5^3 - 1)}{(5 - 1)}$$

More generally, if n = $\prod_{i=1}^{k} p_i^{e_i}$, then the sum of its divisors is $\prod_{i=1}^{k} \frac{p_i^{e_i+1}-1}{p_i-1}$. Note that the capital π is product notation. It's identical to a capital sigma except that it indicates the product of each term instead of the sum of each term.

Prime Factorization of n! (and number of 0s at the end of n!)
Consider the problem of how many times a prime number p divides into n!. n! is simply a listing of each integer from 1 to n, multiplied together. So, naturally, p will divide into p, 2p, 3p, ..., kp, where kp is the largest multiple of p less than or equal to n. More elegantly, we can deduce that k is simply the result of the integer division of n and p. So, p appears at least n/p times in the prime factorization of n!. However, this may not count all of the times p appears in the prime factorization of n!. Consider the term p^2 in the expansion of n! This has the prime factor p twice, but we only counted it once in the count above. But notice that each multiple of p^2 should have counted twice, not once. Thus, we need to add in the integer division of n and p^2. But this only counts two factors of p for the term p^3 (if it exists). Thus, we simply need to continue adding terms until p^k exceeds n. Thus, a general formula for the number of times a prime p divides evenly into n! is $\sum_{i=1}^{\infty} \left\lfloor \frac{n}{p^i} \right\rfloor$. In code, here is a function that accomplishes the calculation:

```
// Pre-condition: p is prime, n > 0.
// Post-condition: returns the number of times p divides into
// n!
public static int numTimesDivide(int n, int p) {
    int res = 0;
    while (n > 0) {
        res += (n/p);
        n /= p;
    }
    return res;
}
```

Notice that a zero at the of a number indicates a divisor of 10 = 2 x 5. Thus, to determine the number of 0s at the end of a number, we simply need to determine the number of times 2 appears in its prime factorization and the number of times 5 appears in its prime factorization. The minimum of these two values equals the number of 0s at the end of the number. Applying this idea to n! and noting that 5 divides into n! fewer times than 2, the number of 0s at the end of n! is equal to the number of times 5 divides into n factorial, since is solved above.

Using Divisibility Arguments to Reduce Brute Force Search

Consider finding all of the positive integer solutions to:

$45x + 1373y = 100000007$

We can consider this equation mod 45 yielding:

$1373y \equiv 100000007 \bmod 45$
$23y \equiv 17 \bmod 45$

Using brute force, we can plug in the possible 45 values for y and see which ones satisfy the equation. Once we know this, then we have a list of all possible values that we can substitute for y, since we can always add or subtract multiples of 45 from the original solution for y and still yield an integer for x.

A better solution to such an equation would be to multiply both sides by the modular inverse of 23 mod 45. By definition, a^{-1} mod n (read: "a inverse mod n") is the integer value, x, (if it exists), in between 0 and n-1 such that $ax \equiv 1 \bmod n$. In our example above, the modular inverse of 23 mod 45 is 2, since $23 \times 2 \equiv 1 \bmod 45$. Thus we can solve the equation above by multiplying it through by 2:

$2(23y) \equiv 2(17) \bmod 45$
$46y \equiv 34 \bmod 45$
$y \equiv 34 \bmod 45$

This is clearly better than trying every value for x or y.

Another example of using a divisibility argument is as follows:

Given an integer, b, representing the length of a leg in a right triangle with integer side lengths, determine all possible values for the other leg and hypotenuse:

Recall that $a^2 + b^2 = c^2$, where a and b are the lengths of the legs of a right triangle and c is the length of the hypotenuse. Now, solve this for b^2:

$b^2 = c^2 - a^2$
$b^2 = (c - a)(c + a)$

It follows that if we're given b, then we must have $(c - a) \mid b^2$.

Since we know that $c - a < c + a$, since a is positive, it also follows that $(c - a) < b$, the square root of b^2.

Thus, our method of solution (so long as the input value of b is no more than 10^9) is to prime factorize b, which then also gives us the prime factorization of b^2. Then, cycle through each divisor, of b^2 that is less than b and set that equal to c - a, creating the following system of equations:

$$c - a = d$$
$$c + a = \frac{b^2}{d}$$

Adding the two equations yields $2c = d + \frac{b^2}{d}$. If the quantity on the right hand side is odd, there's no solution since we require that c is an integer. Otherwise, divide both sides by 2, yielding the value of c and then substitute back for the value of a.

If b $\leq 10^6$, then instead of prime factorizing b, we can just run a brute force search to see which values in between 1 and 10^6 divide evenly into b^2. (Note that based on these bounds we would have to store everything in longs since a million squared does not fit in an int.)